Decomposition During Search for Propagation-Based Constraint Solvers

Martin Mann[1,⋆], Guido Tack[2], and Sebastian Will[1,⋆⋆]

[1] Bioinformatics, Albert-Ludwigs-University Freiburg, Germany
{mmann,will}@informatik.uni-freiburg.de
[2] Programming Systems Lab, Saarland University, Germany
tack@ps.uni-sb.de

Abstract. We describe *decomposition during search* (DDS), an integration of *and/or* tree search into propagation-based constraint solvers. The presented search algorithm dynamically decomposes sub-problems of a constraint satisfaction problem into independent partial problems, avoiding redundant work.
The paper discusses how DDS interacts with key features that make propagation-based solvers successful: constraint propagation, especially for global constraints, and dynamic search heuristics.
We have implemented DDS for the Gecode constraint programming library. Two applications, solution counting in graph coloring and protein structure prediction, exemplify the benefits of DDS in practice.

1 Introduction

Propagation-based constraint solvers owe much of their success to a *best-of-both-worlds* approach: They combine classic AI search methods with advanced implementation techniques from the Programming Languages community and efficient algorithms from Operations Research. Furthermore, the CP community has developed a great number of propagation algorithms for global constraints.

In this paper, we present how to integrate *and/or* search into propagation-based constraint solvers. We call the integration *decomposition during search* (DDS). We take full advantage of all the features mentioned above that make propagation-based constraint solvers successful. The most interesting points, and main contributions of our paper, are how DDS interacts with and benefits from constraint propagation, especially in the presence of global constraints, and dynamic search heuristics.

Related work. Only recently, counting and exhaustive enumeration of solutions of a constraint satisfaction problem (CSP) gained a lot of interest [1,3,10,19]. In general, the counting of CSP solutions is in the complexity class #P, i.e. it is even harder than deciding satisfiability [18]. This class was defined by Valiant [21] as the class of counting problems that can be computed in nondeterministic polynomial time. Notwithstanding the complexity, there is demand for solution counting in real applications. For instance, in bioinformatics counting optimal

⋆ Supported by the EU project EMBIO (EC contract number 012835)
⋆⋆ Supported by the EU Network of Excellence REWERSE (project number 506779)

protein structures is of high importance for the study of protein energy landscapes, kinetics, and protein evolution [22] and can be done using CP [2].

Already folklore, standard solving methods for CSPs like Depth-First Search (DFS) leave room for saving redundant work, in particular when counting *all* solutions [11]. Recent work by Dechter et al. [10,16] introduced *and/or* search for solution counting and optimization, which makes use of repeated *and*-decomposition during the search following a pseudo-tree. Their work thoroughly studies and develops a rich theory of *and/or* trees.

While not in the context of general constraint propagation, similar ideas were discussed before for SAT-solving [3,7,15]. The SAT approaches also introduce the idea of analyzing the induced dependency structure dynamically during the search. This avoids redundancy that occurs due to the emergence of independent connected components in the dependency graph during the search.

Motivation and contribution. The motivation for this paper is to tackle the same kind of redundancy for solving very hard real world problems, such as the counting of protein structures, that require a full-fledged constraint programming system. This requests for a method which is tailored for integration into modern CP systems and directly supports features such as global constraints and dynamic search heuristics. To make use of the statically unpredictable effects of constraint propagation and entailment, the presented method avoids redundant search dynamically.

Our main contribution is to present how to integrate *and/or* tree search into a state-of-the-art, propagation-based constraint solver. We describe *decomposition during search* (DDS) on different levels of abstraction, down to concrete implementation details.

In detail, we stress the impact that constraint propagation has on decomposability of the constraint graph, and how DDS interacts (and works seamlessly together) with propagators for global constraints, the workhorses of modern propagation-based solvers. The practical value of DDS is shown empirically, using a well integrated and competitive implementation for the Gecode library [12].

Overview. The paper starts with a presentation of the notations and concepts that are used throughout the later sections. In Sec. 3, we briefly recapitulate *and/or* search, and then present, on a high level of abstraction, *decomposition during search* (DDS), our integration of *and/or* search into a propagation-based constraint solver. Sec. 4 deals with the interaction of DDS with propagation and search heuristics, and Sec. 5 discusses how global constraints interact with DDS.

On a lower level of abstraction, Sec. 6 sketches the concrete implementation of DDS using the Gecode C++ constraint programming library. With the help of our Gecode implementation, we study the practical impact of DDS in Sec. 7 by counting solutions for random instances of two important CSPs, graph coloring and protein structure prediction. Both examples are hard counting problems (in class #P). The study shows high average speedups and reductions in search tree size, even using our prototype implementation. These two sections therefore provide evidence that DDS can be integrated into a modern constraint program-

ming system in a straightforward way. The paper finishes with a summary and an outlook on future work in Sec. 8.

2 Preliminaries

This section defines the central notions that we want to use to talk about constraint satisfaction problems.

A *Constraint Satisfaction Problem (CSP)* P is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X} = \{x_1, \ldots, x_m\}$ is a finite set of variables, $\mathcal{D}$ a function of variables to their associated value domains, and $\mathcal{C}$ a set of constraints. An n-ary *constraint* $c \in \mathcal{C}$ is defined by the tuple of its n variables $\text{vars}(c)$ and a set of n-tuples of the allowed value combinations. We feel free to interpret $\text{vars}(c)$ as the set of variables of c. A domain $\mathcal{D}$ *entails* a constraint c iff all possible value combinations of the variable domains of $\text{vars}(c)$ in $\mathcal{D}$ are allowed tuples for c. A *solution of a CSP* is an assignment of one value $v \in \mathcal{D}(x_i)$ to each variable $x_i \in \mathcal{X}$ such that all $c \in \mathcal{C}$ are entailed. The *set of solutions of a CSP* P is denoted by $\text{sol}(P)$.

Based on these definitions some important properties of a CSP can be defined. A *CSP* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is solved* iff $\forall x \in \mathcal{X} : |\mathcal{D}(x)| = 1$ and $\text{sol}(P) \neq \emptyset$. *$P$ is failed* iff $\text{sol}(P) = \emptyset$ and *satisfiable* otherwise. A CSP P' is *stronger than* P ($P' \sqsubseteq P$) iff $\text{sol}(P') = \text{sol}(P)$ and $\forall x \in \mathcal{X}' : \mathcal{D}'(x) \subseteq \mathcal{D}(x)$.

The *constraint graph of a CSP* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is a hypergraph $G_P = (V, E)$, where $V = \mathcal{X}$ and $E = \{\text{vars}(c) \mid c \in \mathcal{C}\}$.

3 Decomposition During Search

In this section, we recapitulate *and/or* tree search. Then, we present a high-level model of how to integrate *and/or* tree search into a propagation-based constraint solver. We call this integration *decomposition during search* (DDS).

3.1 *And/or* tree search

Let us look at an example to get an intuition for *and/or* search. Assume $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ with $\mathcal{X} = \{A, B, C, D\}$, $\mathcal{D}(A) = \{3, 5\}$, $\mathcal{D}(B) = \{3, 4\}$, $\mathcal{D}(C) = \mathcal{D}(D) = \{1, 2\}$, and $\mathcal{C} =$ 'A, B, C, D are different'. Figure 1 presents a corresponding search tree for a plain depth-first tree search. Each node is a propagated sub-problem of P and is visualized as a constraint graph. As usual, a node is equivalent to the *disjunction* of all its children.

Even this tiny example demonstrates that plain DFS may perform redundant work: The partial problem on the variables C and D is solved redundantly for each solution of the partial problem on A and B. We say that $\{A, B\}$ and $\{C, D\}$ are *independent* sets of variables.

The central idea of *and/or* tree search [10] is to detect independent partial problems *during search*, to enumerate partial solutions of the partial problems independently, and finally to combine them to solutions, or to compute the number of solutions. That way, each independent partial problem is searched only once. The name *and/or* search stresses that the search tree contains both disjunctive,

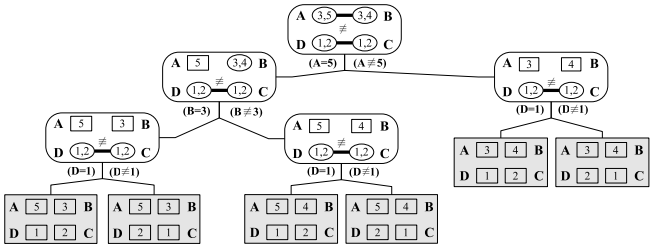


Fig. 1. DFS search tree.

choice nodes and *conjunctive nodes*, representing decompositions. Figure 2 shows a search tree for the same CSP as in Figure 1, but using *and/or* search. For now, you can read the big X as "combine". Here, the search tree contains only one decomposition and two choices, instead of five choices in Figure 1. In general, the amount of redundant work can be exponential in the size of the CSP.

3.2 Integrating *and/or* search into a propagation-based solver

From a bird's eye view, *and/or* search can be done easily in the context of propagation-based constraint solving. Algorithm 1 counts all solutions of a CSP P, decomposing the problem where possible.

Ignoring line 5 for a moment, the algorithm runs a standard depth-first search (DFS). The function PROPAGATE in line 2 performs constraint propagation: it maps a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ to a CSP $P' = (\mathcal{X}, \mathcal{D}', \mathcal{C}')$ such that $P' \sqsubseteq P$. PROPAGATE may remove entailed constraints from $\mathcal{C}$. If P' is failed or solved, we just return that we found no resp. one solution (lines 3,4). Otherwise (line 6), we split the problem into LEFTCHOICE(P') and RIGHTCHOICE(P') (which implement the search heuristic). As the branches correspond to a *disjunction* of P', the recursive counts *add* up to the total number of solutions.

The only addition that is necessary to turn DFS into an *and/or* search is line 5: If the problem can be decomposed into P'_1 and P'_2 (we simplify by assuming only binary decomposition), these partial problems in *conjunction* are equivalent to P'. Hence, we *multiply* the results of the recursive calls.

In contrast to investigating decomposability only on the initial CSP for a static variable selection, Algorithm 1 follows a dynamic approach: the check for decomposability is interleaved with propagation and normal search. As search progresses, more and more variables are assigned to values, and more and more constraints are detected to be entailed. We shall see that this greatly increases the

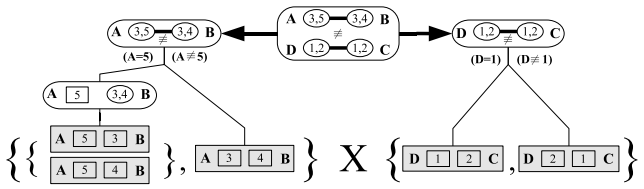


Fig. 2. *And/or* search tree.

Algorithm 1 Counting by Decomposition During Search

```
1: function DDS(P)
2:     P' ← Propagate(P)
3:     if IsFailed(P') return 0
4:     if IsSolved(P') return 1
5:     if Decompose(P')= (P'_1, P'_2) return DDS(P'_1) · DDS(P'_2)        ▷ decomposition
6:     return DDS(LeftChoice(P')) + DDS(RightChoice(P'))                  ▷ choice
7: end function
```

potential for decomposition in Sec. 4. Furthermore, decomposition is completely independent of the implementation of LeftChoice and RightChoice, so any search heuristic can be used.

Short-circuit. As a straight-forward optimization, we can employ short-circuit reasoning in line 5. If $\mathrm{DDS}(P'_1)$ returns no solutions, we do not have to compute $\mathrm{DDS}(P'_2)$ at all. Note the potential pitfall here: There are situations where DFS detects failure easily, but DDS has to search a huge partial problem P'_1 before detecting failure in P'_2. We come back to this in Sec. 4.

Enumerating solutions. Extending DDS to enumeration of solutions is straightforward: We just have to return an empty list in case of failure (line 3), a singleton list with a solution when we find one (line 4), and interpret addition as list concatenation and multiplication as combination of partial solutions. Instead of enumeration, we can also build up a tree-shaped compact representation of the solution space (as in Fig. 2 and later in Fig. 6c).

In the rest of this section, we show how to compute Decompose efficiently.

3.3 Computing the decomposition

We will now define formally when a CSP can be decomposed, and give a sufficient algorithmic characterization that leads to an efficient implementation.

Restriction and independence. The *restriction of a function* $f : \mathcal{Y} \to \mathcal{Z}$ *to a set* $\mathcal{X} \subseteq \mathcal{Y}$ is defined as $f_{|\mathcal{X}} : \mathcal{X} \to \mathcal{Z}, x \mapsto f(x)$.

We define the restriction of a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ to a set of variables $\mathcal{X}' \subseteq \mathcal{X}$ by $P_{|\mathcal{X}'} = (\mathcal{X}', \mathcal{D}_{|\mathcal{X}'}, \mathcal{C}_{|\mathcal{X}'})$, where $\mathcal{C}_{|\mathcal{X}'} = \{c \in \mathcal{C} \mid \mathrm{vars}(c) \subseteq \mathcal{X}'\}$.

A non-empty proper subset $\hat{\mathcal{X}} \subset \mathcal{X}$ is *independent* in a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, iff $\mathrm{sol}(P) = \emptyset$ or $\mathrm{sol}(P)_{|\hat{\mathcal{X}}} = \mathrm{sol}(P_{|\hat{\mathcal{X}}})$. For some $\hat{\mathcal{X}}$ independent in P, we say that $P_{|\hat{\mathcal{X}}}$ is a *partial problem of* P. We can decompose P if it has a partial problem.

The key to an efficient implementation of Decompose is to have an algorithmic interpretation of what it means that a CSP can be decomposed into partial problems. We now show that connected components in the constraint graph of a CSP represent independent partial problems.

A graph is *connected* iff there exists a path between all nodes. A *connected component* of a constraint graph G_P is a maximal connected subgraph.

Proposition. *Consider a CSP* P *with constraint graph* $G_P = (\mathcal{X}, E)$. *If* $\hat{\mathcal{X}} \subset \mathcal{X}$ *is a connected component in* G_P, *then* $P_{|\hat{\mathcal{X}}}$ *is a partial problem of* P. ***Proof:*** *There exists no hyperedge between any node* $x \in \hat{X}$ *and any node* $y \notin \hat{X}$, *as*

connected components are maximal. This means that there is no constraint between such x and y in P. We have to distinguish two cases: If P is unsatisfiable, $\hat{\mathcal{X}}$ is trivially independent (by definition of independence). Otherwise, take an arbitrary solution $s \in \mathrm{sol}(P)$, and an arbitrary solution $s' \in \mathrm{sol}(P_{|\hat{\mathcal{X}}})$. Merging s' into s yields $s'' = (x \mapsto s'(x)$ for $x \in \hat{\mathcal{X}}$, $x \mapsto s(x)$ otherwise). This s'' is again a solution of P, as all constraints on $\mathcal{X} \setminus \hat{\mathcal{X}}$ are still satisfied, and all constraints on $\hat{\mathcal{X}}$ are satisfied, too. As we picked s and s' arbitrarily, we get $\mathrm{sol}(P)_{|\hat{\mathcal{X}}} = \mathrm{sol}(P_{|\mathcal{X}})$.

This result is not new [3,11], but we repeat it to illustrate the central algorithmic idea. Connected components can be computed in linear time in the size of the graph, and incremental algorithms are available. We can thus implement DECOMPOSE as a simple connectedness algorithm on the constraint graph.

Finding more than one non-empty connected component is a sufficient condition for finding partial problems, but not a necessary one. As an example, consider the CSP that contains the trivial constraint allowing all combinations of values for x and y. Then x and y may still be independent, but the constraint graph shows a hyperedge connecting the two variables, so that x and y will always end up in the same connected component. In the following section, we will see how propagation-based solvers can deal with this.

4 How DDS Interacts With Propagation and Search

The previous section showed how DDS can be integrated into a propagation-based solver. But what are the consequences, how is decomposition affected by propagation and search, and how can it benefit from the search heuristic?

4.1 Constraint graph dynamics

Decomposition examines the constraint graph *during search*. This is vital as propagation and search *modify* the constraint graph dynamically – they narrow the domains of the problem's variables and remove some entailed constraints. The result is a sparser constraint graph with more potential for decomposition:

Assignment. Clearly, an assigned variable ($|\mathcal{D}(x)| = 1$) is independent of all other variables of the CSP. This implies that edges to assigned variables can be removed from the constraint graph. Variables can get assigned during search and by propagation, so both mechanisms result in sparser constraint graphs.

Entailment. If a constraint c is entailed in a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, it will not contribute to propagation any more. Formally, we have for $P' = (\mathcal{X}, \mathcal{D}, \mathcal{C} \setminus \{c\})$ that $\mathrm{sol}(P') = \mathrm{sol}(P)$. It is also obvious that the constraint graph for P' is sparser than the one for P, it contains one edge less. It is thus vital to our approach to detect entailment of constraints as early as possible, and to remove entailed constraints from $\mathcal{C}$. Clearly, full entailment detection is coNP-complete. Most CP systems however implement a weak form of entailment detection in order to remove propagators early, which our approach automatically benefits from.

4.2 Search heuristics

The search heuristic, encoded by LeftChoice and RightChoice, is extremely important for the efficiency of the search. In particular, dynamic heuristics, natively supported by DDS, are known to be largely superior to static ones.

A common class of search heuristics for solving CSPs are *variable-value heuristics*. Such a binary heuristic selects a variable x and a value v from this variable's domain. The selection may depend on the current sub-problem. For a variable-value heuristic, LeftChoice(P) returns P with the added constraint $x \diamond v$, and RightChoice(P) adds $\neg(x \diamond v)$ to P, where $\diamond$ is some binary relation.

Variable selection. The variable selection strategy of such a heuristic is crucial for the size of the search tree. A common method for variable selection is 'first-fail', which selects by minimal domain size. Other strategies use the degree in the constraint graph or the minimal/maximal/median value in the domain. Static variable orderings are in many cases inferior to these dynamic strategies.

Heuristics that maximize decomposition. In order to maximize the number of decompositions during search, the heuristic can be guided by the constraint graph. Such a heuristic may compute e.g. cut-points, bridges, or more powerful (minimal) cut-sets/separators. Our framework is well prepared to accommodate such complex strategies, in particular because our method already builds on access to the constraint graph.

Order of exploration of partial problems. The exploration order of *and*-nodes, the partial problems, is of high importance. After detecting inconsistency in one partial problem, exploration of the other ones is needless. Because of the duality of *and*- and *or*-nodes, we can reuse the *or*-heuristic for decomposition. Assuming that we already have a good variable selection heuristic, we use this heuristic to guide the partial problem ordering: choose the partial problem that contains the selected variable first. That way, a good first-fail heuristic will lead to failure in the first explored partial problem. This mitigates the effect that failure may be detected late using DDS.

5 Global Constraints

One of the key features of modern constraint solvers is the use of global constraints to strengthen propagation. Therefore, a search algorithm has to support global constraints in order to be practically useful in such systems.

For an n-ary (global) constraint, the constraint graph contains a hyperedge that connects all n variable nodes. Considering the example from Fig. 1 again, we introduce one all-different instead of all binary $\neq$-constraint. The domain-consistent propagator for this constraint is clearly at a fixpoint. Furthermore, the variable sets $\{A, B\}$ and $\{C, D\}$ are independent concerning all-different. However, the constraint graph contains just one hyperedge between all four variables. Thus, in the current set-up, the all-different constraint *prevents decomposition*.

In the following, we will show how to deal with this problem in general, and how we can efficiently compute decompositions for three standard global

constraints. We discuss global constraints for linear (in-)equations and why such equations hinder decomposition. Finally, we generalize the examples to derive a general rule that tells if a global constraint is decomposable or not.

Decomposing global constraints. In general, most global constraints can be thought of as efficient implementations for a conjunction of smaller constraints [6]. This is usually also called the *decomposition* of the global constraint – we will call it *constraint decomposition* to distinguish it from the *constraint graph decomposition* we are interested in.

The constraint decomposition of a global constraint gives us the tool for computing the constraint graph decomposition in the presence of global constraints. Instead of the original graph, we use the graph containing the constraint decomposition (or an approximation) to compute connected components.

For the introductory example of the all-different constraint, this would mean that we consider exactly the connected components shown in Fig. 2, yielding the same decomposition.

5.1 All-different

Régin's propagator for the all-different constraint [20] employs a bipartite graph, the *variable-value* graph. It connects each variable node with the value nodes corresponding to the elements of the current variable domain. We can observe that all-different can be decomposed exactly if the variable-value graph contains more than one connected component (see Fig. 3 a). As connected components of this graph have to be computed anyway during the propagation, we can get the constraint decomposition without any additional overhead. This technique generalizes to the **global cardinality constraint**.

5.2 Slide

This constraint, introduced by Bessiere et al. [4], slides a k-ary constraint c over a sequence of variables, i.e. it holds if $c(x_i, \ldots, x_i + k - 1)$ holds for all $1 \leq i \leq n - k + 1$. The constraint can be split into two at variable x_d if the individual constraints involving x_d are entailed (see Fig. 3 b). Entailment happens at the latest when all variables between x_{d-k+1} and x_{d+k-1} are assigned. Depending on how soon the individual c are entailed, we can decompose even earlier.

5.3 Regular

Pesant's regular language membership constraint [17] states that the values taken by a sequence of variables $x_1, \ldots, x_n$ belong to a regular language L. The propagation algorithm works on the so-called *layered graph*, which represents an unfolding of a finite automaton accepting L (see Fig. 3 c).

If now, at some point during propagation, one layer is left with a single state (see Fig. 3 c), the graph can be split into two halves, making the singleton state a new final state (for the left half) and start state (for the right half). They correspond to regular expressions R_l and R_r, covering the two substrings left and right of that layer, such that the language generated by $R_l R_r$ is a sublanguage of L that contains exactly those strings still licensed by the variable domains. Note, the constraint decomposition is possible even without variable assignment.

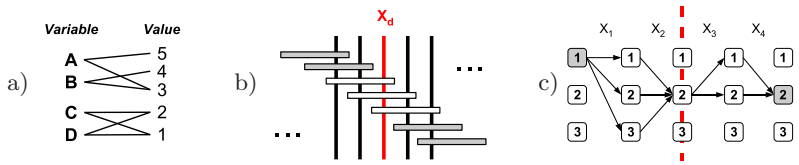


Fig. 3. (a) Variable-value graph of an All-different constraint for the CSP in Fig. 1, (b) Decomposing the slide constraint, (c) Layered graph for a regular constraint.

5.4 Linear (in-)equations

Linear (in-)equations are ubiquitous in real-world constraint problems. Unfortunately, it turns out that they hinder decomposition. Consider the simple linear equation $\sum_{i=1}^{n} x_i = c$. It can be decomposed into ternary equations $x_1 + x_2 = y_1$, $x_2 + y_1 = y_2$, $\ldots$, $x_n + y_{n-1} = c$. The corresponding constraint graph decomposes if one of the y_i is assigned. However, this only happens when all the x_j to the right or all the x_j to the left of y_i are assigned. Thus, constraint subgraphs with linear constraints will stay connected until the variables are assigned.

5.5 Decomposability of global constraints

Revisiting the four examples, a general rule for the decomposability of global constraints can be derived. The all-different, slide and regular constraints have rather different propagation algorithms. Nevertheless, the constraints share a common feature: there exists a decomposition without additional variables. This is not the case for linear (in-)equations and related constraints. Here, intermediate variables are necessary that hinder the constraint decomposition.

Decomposability of global constraints is an active area of research [6,4,5]. It makes for interesting future work to analyze constraint decompositions with respect to their impact on DDS.

6 Implementation

Our implementation of DDS extends Gecode, a C++ constraint programming library. In this section, we give an overview of relevant technical details of Gecode, and discuss the four main additions to Gecode that enable DDS: access to the constraint graph, decomposing global constraints, integrating Decompose into the search heuristic, and specialized search engines. The additions to Gecode comprise only 2500 lines (5%) of C++-code and enable the use of DDS in any CSP modeled in Gecode. The source code of the prototype implementation is available from the authors upon request.

Gecode. The Gecode library [12] is an open source constraint solver implemented in C++. It lends itself to a prototype implementation of DDS because of four facts: (1) Full source code enables changes to the available propagators. (2) The reflection capabilities allow access to the constraint graph. (3) Search is based on recomputation and copying, which significantly eases the implementation of specialized branchings and search engines. (4) It provides good performance, so that benchmarks give meaningful results.

Constraint graph. In most CP systems, the constraint graph is implicit in the data structures for variables and propagators. Gecode, e.g., maintains a list of propagators, and each propagator has access to the variables it depends on.

For DDS, a more explicit representation is needed that supports the computation of connected components. We can thus either maintain an additional, explicit constraint graph during propagation and search, or extract the graph from the implicit information each time we need it. For the prototype implementation, we chose the latter approach. We make use of Gecode's reflection API, which allows to iterate over all propagators and their variables. Through reflection, we construct a graph using data structures from the boost graph library [8], which also provides the algorithm that computes connected components.

Global constraints. There are two possible implementation strategies for decomposing global constraints as discussed in Sec. 5. First, a propagator detects its decomposability during propagation and replaces itself with several smaller propagators. For the second strategy, the decomposition is only computed lazily, when the constraint graph is accessed. We implemented the latter option.

Decomposing branchings. Once we have identified connected components in the constraint graph, we have to create the partial problems that correspond to these components. In Gecode, we exploit the duality of choice and decomposition: both add branches to the search tree. The following observation leads to a simple and efficient implementation. If the heuristic is restricted such that it only selects variables inside one connected component, also propagation will only occur for variables of that component: For $\hat{\mathcal{X}}$ independent in P, $\text{propagate}(P)_{|\hat{\mathcal{X}}} = \text{propagate}(P_{|\hat{\mathcal{X}}})$.

For our Gecode implementation, DECOMPOSE is thus realized as a *branching*. A branching in Gecode usually implements LEFTCHOICE/RIGHTCHOICE. For DDS, we extend it to also implement DECOMPOSE: If decomposition is possible, the branching limits further search to the variables in one connected component per branch. Otherwise, it just creates the usual choices according to the heuristic.

Branchings in Gecode are fully programmable. They have to support two operations[3]: DESCRIPTION and COMMIT. DESCRIPTION returns an abstract description of the possible branches while COMMIT executes the branching according to a given description and alternative number.

A decomposing branching in Gecode is a wrapper around a standard variable-value branching. The actual work is done by DESCRIPTION: it requests the constraint graph and performs the connected component analysis. If decomposition is possible, a special description is returned, representing the independent subsets $\hat{\mathcal{X}}_i \subset \mathcal{X}$. Otherwise, DESCRIPTION is delegated to the embedded variable-value branching. Note that Gecode supports n-ary branchings, so decompositions do not have to be binary (as presented so far).

When COMMIT is invoked with a variable-value description, the call is again delegated to the embedded branching. For a decomposition description, the

[3] In fact, branchings in Gecode have a slightly more complicated interface, which we deviate from to simplify presentation.

branching's list of variables is updated to $\hat{\mathcal{X}}_i$ for branch i, those still active in the selected component.

Decomposition search engines.
As decomposition is performed by the branching, the search engines have to be specialized accordingly. We developed four search engines for DDS. A counting search engine computes the number of solutions of a given problem. A general-purpose search engine allows to incrementally search the whole tree and access all the partial solutions. Based on that we provide a search engine that enumerates all full solutions. A graphical search engine based on Gecode's *Gist* (graphical interactive search tool) displays the search tree with special decomposition nodes, and allows to get an overview of where and how a particular problem can be decomposed. Figure 4 shows a screen shot of a partial search using DDS. Circular nodes with inner squares represent decompositions. All search engines accept cut-off parameters for the number of (full, not partial!) solutions to be explored.

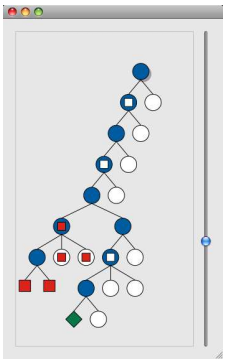


Fig. 4. Gist

7 Applications and Empirical Results

To illustrate possible use cases of DDS we applied it to two counting problems with global constraints, graph coloring and optimal protein structure prediction. Both applications show tremendous reductions in runtime and search tree size.

The applications were realized using our DDS implementation in Gecode. Only the search strategy was changed (DFS to DDS) – modeling, variable and value selection were kept the same for an appropriate comparison of the results. We chose maximal degree with minimal domain size as tie breaker as dynamic variable selection, which enforces decomposition and works well for DFS, too.

7.1 Graph coloring

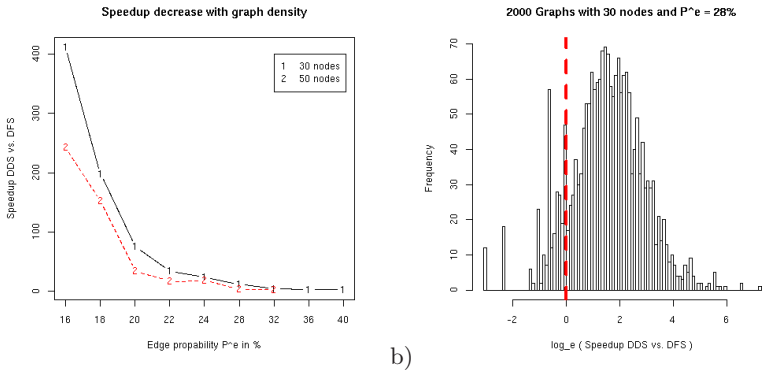
Graph coloring is an important and hard problem with applications in scheduling, assignment of radio frequencies, and computer optimization. A proper coloring assigns different colors to adjacent nodes. We want the chromatic polynomial for the chromatic number, i.e. the number of graph colorings with minimal colors.

The constraint model. For a given undirected graph g and a number of colors c we introduce one variable per node with the initial domains $0..(c-1)$. For each maximal clique of size > 2, we post an all-different constraint on the corresponding variables. For all remaining edges we add binary disequality constraints.

The test sets. We generated the two test sets GC-30 and GC-50 of graphs with 30 and 50 nodes. For each size, random graphs were obtained by inserting an edge of the complete graph with a fixed uniform edge probability P^e. This was done using the Erdős-Rényi random graph generator GTgraph [13]. For each edge probability P^e from 16 to 40 percent, 2000 graphs were generated and their

DFS/DDS:	Test set	16 %	18 %	20 %	22 %	24 %	28 %	32 %	36 %	40 %
Rel. runtime:	GC-30	411.2	197.7	75.74	34.6	23.1	11.9	3.85	2.74	2.14
	GC-50	242.7	151.8	34.23	16.5	18.2	3.4	2.71	–	–
Search tree size:	GC-30	680.3	344.4	142.0	74.48	62.27	33.96	10.90	6.86	4.97
	GC-50	646.1	383.8	94.28	47.26	41.69	11.6	9.28	–	–

Table 1. Average ratios of DFS vs. DDS for various edge probabilities



a) b)

Fig. 5. (a) Avg. speedup decrease from 400 to 2 by graph density, (b) Histogram of logarithmic speedup for $P^e = 28$ and 30 nodes (the dashed line marks equal runtime).

colorings counted via DFS and DDS. To handle highly degenerated problems as well, we stopped after 1 million solutions.

Results. For the test sets, Tab. 1 compares the time consumption and search tree size by average ratios of DFS and DDS ($\frac{DFS}{DDS}$). A figure of 100 thus means that DDS is 100 times faster than DFS, or that the DFS search tree has 100 times as many nodes as the one for DDS. A dash means that most of the problems were not solved within a given time-out.

The presented runtime ratios show the high speedup for graphs with edge probabilities $P^e \leq 40\%$. The distribution of speedup is exemplified in Fig. 5b. The speedup corresponds to an even larger reduction of the search tree for DDS, which was only increased for 0.5% of all problems. Furthermore, sparse graphs yield a much higher runtime improvement than dense graphs, visualized by Fig. 5a. The number of fails and propagations show no significant effect of DDS in contrast to runtime or search tree size.

Still, the search tree reduction is not completely reflected in runtime speedup, which illustrates the computational overhead of DDS in the current prototypic implementation. Anyway, our data shows that DDS is well suited to improve solution counting even for dense graphs with P^e about 40%. We expect even higher speedups and search-tree reductions if the solutions are counted completely, i.e. without the current upper bound of 1 million. Table 1 suggests that the speedup decreases with increasing number of nodes to color in the graph. With increasing number of nodes, the graph *as well as the constraint graph* grow quadratically.

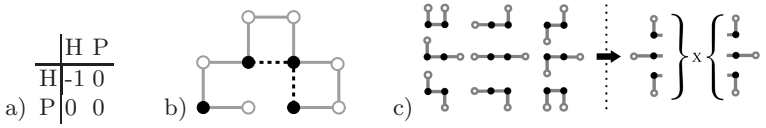


Fig. 6. (a) Contact energy function (b) Example HP-structure with energy -2 (H-monomer: black, P-monomer: white, structure back bone: grey, HH-contact: dotted) (c) Compression of the solution space from 9 complete down to 3 by 3 partial structures.

The speedup is significantly lower than the reduction of the search tree. In part this can be ascribed to our implementation that rebuilds the constraint graph in each search step. A system that provides cheaper constraint graph access, e.g. by maintaining it incrementally, is expected to perform much better.

7.2 Optimal protein structure prediction

The prediction of optimal (minimal energy) structures of simplified lattice proteins is a hard (NP-complete) problem in bioinformatics. Here we focus on the HP-model introduced in [14]. In this model, a protein chain is reduced to a sequence of monomers of equal size, whereby the 20 aminoacids are grouped into hydrophobic (H) or polar (P). A structure is a self-avoiding walk of the underlying lattice (e.g. square or cubic). A contact energy function is used to determine the energy of a structure. The energy table and an example is given in Fig. 6. The problem is to predict minimal energy structures for a given HP-sequence.

The number and quality of optimal structures has applications in the study of energy landscape properties, protein evolution and kinetics [22].

The constraint model. In [2], the problem was successfully modeled as CSP and named Constraint-based Protein Structure Prediction (CPSP). Here, a variable is introduced for each sequence position and with lattice points as domains[4]. The self-avoiding walk is modeled by a sequence of binary neighboring constraints (ensuring the connectivity of successive monomers) and a global all-different constraint for self-avoidingness. Supporting decomposition of the all-different propagator, see Sec. 5, is therefore essential for profiting from DDS.

CPSP uses a database of pre-calculated point sets, called H-cores, that represent possible optimal distributions of H-monomers. By that, the optimization problem is reduced to a satisfaction problem for a given H-core, if H-variables are restricted to these positions. For optimal H-cores, the solutions of the CSP are optimal structures. Thus, for counting all optimal structures, one iterates through the optimal cores.

The test sets. We generated two test sets, PS-48 and PS-64, with uniformly distributed random HP-sequences of length 48 and 64. For the generation we used the free available CPSP implementation [9]. With only minimal modifications (new branching) we use the existing CSP model with DDS.

[4] In practice, lattice positions are indexed by integers such that standard constraint solvers for finite domains over integers are applicable.

	DFS / DDS			
	runtime	search tree size	fails	propagations
PS-48	2.98	11.30	1.40	3.27
PS-64	4.23	25.33	1.76	5.43

Table 2. Average ratios for CPSP using DFS vs. DDS (ST = search tree)

PS-48 contains 6350 HP-sequences and for each up to 1 million optimal structures in the cubic lattice were predicted. For the 2630 HP-sequences in PS-64 up to 2 million structures have been predicted in the cubic lattice, due to the increasing degeneracy in sequence length.

Results. The average ratio results are given in Tab. 2. There, the enormous search tree reduction with an average factor of 11 and 25 respectively is shown. The reduction using DDS compared to DFS leads to much less propagations (3- to 5-fold). This and the slightly less fails result in a runtime speedup of 3-/4-fold using the same variable selection heuristics for both search strategies. Here, the immense possibilities of DDS even without advanced constraint-graph specific heuristics are demonstrated. This also shows the rising advantage of DDS over DFS for increasing problem sizes (with higher solution numbers).

8 Discussion

Our paper introduces *decomposition during search* (DDS), an integration of *and/or* search with propagation-based constraint solvers. DDS dynamically decomposes CSPs, avoiding much of the redundant work of standard tree search when exploring huge search spaces, e.g. of $\#P$-hard counting problems.

Building on folklore knowledge and previous research, we discuss the interaction of DDS with such vital features as global constraints and dynamic variable ordering. The techniques presented here have been implemented for Gecode.

Our empirical evaluation on graph coloring and protein structure prediction shows the huge potential of DDS in terms of search tree size reduction and already high true runtime speedup. The speedup proves that DDS can be implemented competitively, and with a reasonable overhead. We expect even higher speedups by improving the constraint graph representation and its incremental maintenance, which is a current area of development. However, one experience from our experiments is that it is highly problem-specific whether the constraint graph allows for decomposition. We partly explain this by pointing out that some constraints (e.g. linear (in-)equations) inherently hinder decomposition.

We envision promising future research in the following directions. First, providing efficient access to the constraint graph. Second, the development of specifically tailored heuristics for DDS focusing on dynamic variable selection or domain splitting. Such heuristics should employ information about the constraint graph, to decompose the problem as often as possible and in a well-balanced way. Decomposition-directed heuristics might however counteract problem specific heuristics. Balancing such heuristics is a further research direction.

Finally, solving optimization problems using an *and/or* branch-and-bound (BAB) search [16] seems an obvious extension. However, our first experiments using a prototypical DDS extension of BAB show much smaller benefits than for counting (similar to the results in [16]).

Acknowledgements. We thank Christian Schulte and Mikael Lagerkvist for fruitful discussions about the architecture and the paper, and the reviewers of an earlier version of this paper for constructive comments.

References

1. Ola Angelsmark and Peter Jonsson. Improved algorithms for counting solutions in constraint satisfaction problems. In *Proc. of 9th CP*, 2003.
2. Rolf Backofen and Sebastian Will. A constraint-based approach to fast and exact structure prediction in 3d protein models. *Constraints*, 11(1):5–30, 2006.
3. R. J. Bayardo Jr. and J. D. Pehoushek. Counting models using connected components. In *Proc. of the 7th National Conference on AI*, 2000.
4. Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, Claude-Guy Quimper, and Toby Walsh. Reformulating global constraints: the SLIDE and REGULAR constraints. In *Proc. of SARA-2007*, 2007.
5. Christian Bessière, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Disjoint, partition and intersection constraints for set and multiset variables. In *Proc. of 10th CP*, 2004.
6. Christian Bessière and Pascal Van Hentenryck. To be or not to be ... a global constraint. In *Proc. of 9th CP*, 2003.
7. Armin Biere and Carsten Sinz. Decomposing SAT problems into connected components. *Satisfiability, Boolean Modeling and Computation*, 2:191–198, 2006.
8. Boost graph library, 2007. Available as an open-source library from `www.boost.org`.
9. CPSP: Tools for constraint-based protein structure prediction, 2006. Available as an open-source library from `www.bioinf.uni-freiburg.de/sw/cpsp`.
10. Rina Dechter and Robert Mateescu. The impact of AND/OR search spaces on constraint satisfaction and counting. In *Proc. of 10th CP*, 2004.
11. Eugene C. Freuder and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proc. of 9th IJCAI*, 1985.
12. Gecode: Generic constraint development environment, 2007. Available as an open-source library from `www.gecode.org`.
13. GTgraph: A suite of synthetic graph generators, 2006. Available as an open-source library from `www-static.cc.gatech.edu/~kamesh/GTgraph`.
14. Kit Fun Lau and Ken A. Dill. A lattice statistical mechanics model of the conformational and sequence spaces of proteins. *ACS*, 22:3986 – 3997, 1989.
15. Wei Li and Peter van Beek. Guiding real-world SAT solving with dynamic hypergraph separator decomposition. In *Proc. of 16th IEEE ICTAI*, 2004.
16. Radu Marinescu and Rina Dechter. AND/OR branch-and-bound for graphical models. In *Proc. of 19th IJCAI*, 2005.
17. Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *Proc. of 10th CP*, 2004.
18. Gilles Pesant. Counting solutions of CSPs: A structural approach. In *Proc. of 19th IJCAI*, 2005.
19. Dan Roth. On the hardness of approximate reasoning. *Artif. Intelligence*, 82(1-2):273–302, 1996.
20. Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of 12th National Conference on AI*, pages 362–367, 1994.
21. Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.
22. M. Wolfinger, S. Will, I. Hofacker, R. Backofen, and P. Stadler. Exploring the lower part of discrete polymer model energy landscapes. *EPL*, 74:725–732, 2006.